**Searching Algorithms:**

**Binary search**

#include <iostream>

#include<conio.h>

using namespace std;

int main()

{

int i, num, first, last, middle, n;

cout <<"\n Enter number of elements in array:";

cin >>n;

int \* arr = new int(n);

cout << "\n Enter "<<n<< " elements in ascending order: ";

for(i=0; i<n; i++)

{

cout << "\n Enter the value for "<< i+1<< ": ";

cin>>arr[i];

}

cout<< "\n Enter the element to be searched: ";

cin>>num;

first = 0;

last = n - 1;

middle = (first + last) /2;

while (first <= last)

{

if(arr [middle] < num)

first = middle + 1;

else if (arr [middle] == num)

{

cout << "\n The number "<< num << " is found at position " << middle + 1;

break;

}

else

last = middle - 1;

middle = (first + last ) /2;

}

if (first>last)

cout << "\n The number "<< num <<" is not found in given array !";

getch();

return 0;

}

**Linear search**

#include <iostream>

#include <conio.h>

using namespace std;

int main()

{

int i,n,arr[50],s,flag=1;

cout<<"Enter the size of array: ";

cin >>n;

cout<< "\n Enter the value in ascending order:";

for(i=0; i<n; i++)

{

cout << "\n Enter the value for "<< i+1<< ": ";

cin>>arr[i];

}

cout <<" \n Enter the number you want to search:";

cin >>s;

for(i=0;i<n;i++)

{

if (arr [i] == s)

{

cout << "SEARCH SUCCESSFUL";

cout << "The given number at: "<<i+1;

flag=0;

break;

}

}if(flag ==1)

{cout<<"SEARCH UNSUCCESSFUL";

} getch();

return 0;

}

**Fibonacci search**

#include<iostream>

#include<conio.h>

using namespace std;

int get\_fib(int n) {

int fibk = 0;

int fibk2 = 0;

int fibk1 = 1;

while (fibk < n) {

fibk = fibk1 + fibk2;

fibk2 = fibk1;

fibk1 = fibk;

}

return fibk2;

}

int main() {

int i, n, key, index, low, high, location, flag;

cout << "Enter the number of elements in array : ";

cin >> n;

int \*arr = new int[n];

cout << "\nEnter " << n << " elements in ascending order: ";

for (i = 0; i < n; i++)

cin >> arr[i];

low = 0;

location = -1;

high = n - 1;

flag = 0;

cout << "\nEnter the element to be searched: ";

cin >> key;

index = 0;

while (flag != 1 && low <= high) {

index = get\_fib(high - low + 1);

if (key == arr[index + low]) {

location = low + index;

flag = 1;

break;

} else if (key > arr[index + low]) {

low = low + index + 1;

} else {

high = low + index - 1;

}

}

if (flag == 1)

cout << "Element is found at the position " << location + 1;

else

cout << "Element is not found in the array!";

getch();

return 0;

}

**Evaluation of Arithmetic Expressions**

import java.util.\*;

public class Main{

public static void main(String[] args){

Scanner scan =new Scanner(System.in);

System.out.println("enter the expression");

String input =scan.next();

if(Character.isDigit(input.charAt(0)) && Character.isDigit(input.charAt(input.length()-1)) ||

input.charAt(0)=='('||

input.charAt(input.length()-1)==')')

{

infix(input);

}

else if(Character.isDigit(input.charAt(0)) && Character.isDigit(input.charAt(1))){

System.out.println("it is postfix"+postfix(input));

}

else if(Character.isDigit(input.charAt(input.length()-1)) && Character.isDigit(input.charAt(input.length()-2)))

{

System.out.println("it is prefix"+prefix(input));

}

}

public static float prefix(String input){

Stack<Float> stk=new Stack<>();

int size=input.length()-1;

for(int i=size;i>=0;i--){

if(Character.isDigit(input.charAt(i))){

stk.push((float)Character.getNumericValue(input.charAt(i)));

}

else{

float o1=stk.peek();

stk.pop();

float o2=stk.peek();

stk.pop();

if(input.charAt(i)=='+'){

stk.push(o1+o2);

}

else if(input.charAt(i)=='-'){

stk.push(o1-o2);

}

else if(input.charAt(i)=='\*'){

stk.push(o1\*o2);

}

else if(input.charAt(i)=='/'){

stk.push(o1/o2);

}

else if(input.charAt(i)=='^'){

stk.push((float)(Math.pow(o1,o2)));

}

else{

System.out.println("invalid operator");

return -1;

}

}

}

return stk.peek();

}

public static float postfix(String input){

Stack<Float> stk=new Stack<>();

int size=input.length()-1;

for(int i=0;i<=size;i++){

if(Character.isDigit(input.charAt(i))){

stk.push((float)Character.getNumericValue(input.charAt(i)));

}

else{

float o2=stk.peek();

stk.pop();

float o1=stk.peek();

stk.pop();

if(input.charAt(i)=='+'){

stk.push(o1+o2);

}

else if(input.charAt(i)=='-'){

stk.push(o1-o2);

}

else if(input.charAt(i)=='\*'){

stk.push(o1\*o2);

}

else if(input.charAt(i)=='/'){

stk.push(o1/o2);

}

else if(input.charAt(i)=='^'){

stk.push((float)(Math.pow(o1,o2)));

}

else{

System.out.println("invalid operator");

return -1;

}

}

}

return stk.peek();

}

public static void infix(String input){

Stack<Character> stk= new Stack<>();

String post="";

for(int i=0; i<input.length();i++){

char c = input.charAt(i);

if(c>='0' && c<='9')

{

post+=c;

}

else if(c=='('){

stk.push('(');

}

else if(c==')'){

while(stk.peek() != '('){

post+=stk.peek();

stk.pop();

}

stk.pop();

}

else{

while(!stk.empty() && prec(input.charAt(i)) <= prec(stk.peek())){

post+=stk.peek();

stk.pop();

}

stk.push(c);

}

}

while(!stk.empty()){

post+=stk.peek();

stk.pop();

}

System.out.println("the post fix of give exp.." + post);

System.out.println(postfix(post));

}

public static int prec(char c){

if(c=='^'){

return 3;}

else if(c=='/' || c=='\*'){

return 2;}

else if(c=='+' || c=='-'){

return 1;}

else{

return -1;

}

}

}

**Stack**

#include <iostream>

using namespace std;

#define N 10

int top=-1;

int stack[N];

void push();

void pop();

void stop();

void isempty();

void isfull();

int main(){

int opp=0;

do{

cout<<"Enter the option:\n1.push\n2.pop\n3.top\n4.isempty\n5.isfull\n6.exit\n";

cin>>opp;

switch (opp)

{

case 1:

push();

break;

case 2:

pop();

break;

case 3:

stop();

break;

case 4:

isempty();

break;

case 5:

isfull();

break;

}

}while (opp!=6);

}

void push(){

if(top==N-1){

cout<<"stack is full...";

}

else{

cout<<"Enter the element to be pushed.";

top++;

cin>>stack[top];

cout<<"the element is pushed.\n";

}

}

void pop(){

if(top==-1){

cout<<"stack is empty...";

}

else{

cout<<"the element to be poped."<<stack[top];

top--;

}

}

void stop(){

if(top==-1){

cout<<"stack is empty...";

}

else{

cout<<"top of the stack."<<stack[top];

}

}

void isempty(){

if(top==-1){

cout<<"stack is empty...";

}

else{

cout<<"Stack is not empty...";

}

}

void isfull(){

if(top==N-1){

cout<<"stack is full...";

}

else{

cout<<"Stack is not full...";

}

}

**Queue**

#include <iostream>

using namespace std;

int queue[100], n = 100, front = - 1, rear = - 1;

void Insert() {

int val;

if (rear == n - 1)

cout<<"Queue Overflow"<<endl;

else {

if (front == - 1)

front = 0;

cout<<"Insert the element in queue : "<<endl;

cin>>val;

rear++;

queue[rear] = val;

}

}

void Delete() {

if (front == - 1 || front > rear) {

cout<<"Queue Underflow ";

return ;

} else {

cout<<"Element deleted from queue is : "<< queue[front] <<endl;

front++;;

}

}

void frontelement(){

if (front == - 1 || front > rear)

cout<<"Queue Empty";

else

cout<<"Element in the front is : "<< queue[front] <<endl;

}

void Display() {

if (front == - 1)

cout<<"Queue is empty"<<endl;

else {

cout<<"Queue elements are : ";

for (int i = front; i <= rear; i++)

cout<<queue[i]<<" ";

cout<<endl;

}

}

int main() {

int ch;

cout<<"1) Insert element to queue"<<endl;

cout<<"2) Delete element from queue"<<endl;

cout<<"3) Front Element in the queue"<<endl;

cout<<"4) Display all the elements of queue"<<endl;

cout<<"5) Exit"<<endl;

do {

cout<<"Enter choice : "<<endl;

cin>>ch;

switch (ch) {

case 1: Insert();

break;

case 2: Delete();

break;

case 3: frontelement();

break;

case 4: Display();

break;

case 5: cout<<"Exitting the program !"<<endl;

break;

default: cout<<"Invalid choice"<<endl;

}

} while(ch!=5);

return 0;

}

**Circular** **queue**

#include <iostream>

using namespace std;

int cqueue[5];

int front = -1, rear = -1, n=5;

void insertCQ(int val) {

if ((front == 0 && rear == n-1) || (front == rear+1)) {

cout<<"Queue Overflow \n";

return;

}

if (front == -1) {

front = 0;

rear = 0;

} else {

if (rear == n - 1)

rear = 0;

else

rear = rear + 1;

}

cqueue[rear] = val ;

}

void deleteCQ() {

if (front == -1) {

cout<<"Queue Underflow\n";

return ;

}

cout<<"Element deleted from queue is : "<<cqueue[front]<<endl;

if (front == rear) {

front = -1;

rear = -1;

} else {

if (front == n - 1)

front = 0;

else

front = front + 1;

}

}

void displayCQ() {

int f = front, r = rear;

if (front == -1) {

cout<<"Queue is empty"<<endl;

return;

}

cout<<"Queue elements are :\n";

if (f <= r) {

while (f <= r){

cout<<cqueue[f]<<" ";

f++;

}

} else {

while (f <= n - 1) {

cout<<cqueue[f]<<" ";

f++;

}

f = 0;

while (f <= r) {

cout<<cqueue[f]<<" ";

f++;

}

}

cout<<endl;

}

int main() {

int ch, val;

cout<<"1)Insert\n";

cout<<"2)Delete\n";

cout<<"3)Display\n";

cout<<"4)Exit\n";

do {

cout<<"Enter choice : "<<endl;

cin>>ch;

switch(ch) {

case 1:

cout<<"Input for insertion: "<<endl;

cin>>val;

insertCQ(val);

break;

case 2:

deleteCQ();

break;

case 3:

displayCQ();

break;

case 4:

cout<<"Exit\n";

break;

default: cout<<"Incorrect!\n";

}

} while(ch != 4);

return 0;

}

**Priority queue**

#include<iostream>

#define N 20

using namespace std;

int Q[N],Pr[N];

int r = -1,f = -1;

void enqueue(int data,int p)

{

int i;

if((f==0)&&(r==N-1))

cout<<"Queue is full";

else

{

if(f==-1)

{

f = r = 0;

Q[r] = data;

Pr[r] = p;

}

else if(r == N-1)

{

for(i=f;i<=r;i++) {

Q[i-f] = Q[i];

Pr[i-f] = Pr[i];

r = r-f;

f = 0;

for(i = r;i>f;i--)

{

if(p>Pr[i])

{

Q[i+1] = Q[i];

Pr[i+1] = Pr[i];

}

else

break;

Q[i+1] = data;

Pr[i+1] = p;

r++;

}

}

}

else

{

for(i = r;i>=f;i--)

{

if(p>Pr[i])

{

Q[i+1] = Q[i];

Pr[i+1] = Pr[i];

}

else

break;

}

Q[i+1] = data;

Pr[i+1] = p;

r++;

}

}

}

void print()

{

int i;

for(i=f;i<=r;i++)

{

cout<<"Element = "<<Q[i]<<" Priority = "<<Pr[i]<<endl;

}

}

int dequeue()

{

if(f == -1)

{

cout<<"Queue is Empty";

}

else

{

cout<<"Deleted Element = "<<Q[f]<<endl;

cout<<"It's Priority = "<<Pr[f]<<endl;

if(f==r)

f = r = -1;

else

f++;

}

}

int main()

{

int opt,n,i,data,p;

cout<<"Enter Your Choice:-"<<endl;

do{

cout<<"1) Insert the Data in Queue\n2) Show the Data in Queue \n3) Delete the data

from the Queue\n0) Exit"<<endl;

cin>>opt;

switch(opt){

case 1:

cout<<"Enter the number of data"<<endl;

cin>>n;

cout<<"Enter your data and Priority of data"<<endl;

i=0;

while(i<n){

cin>>data;

cin>>p;

enqueue(data,p);

i++;

}

break;

case 2:

print();

break;

case 3:

dequeue();

break;

case 0:

cout<<"Exitting the program !";

break;

default:

cout<<"Incorrect Choice"<<endl;

}

}while(opt!=0);

 return 0;

}

**Linked List:**

**Singly Linked List**

#include<iostream>

using namespace std;

class Node {

public:

int key;

int data;

Node \* next;

Node() {

key = 0;

data = 0;

next = NULL;

}

Node(int k, int d) {

key = k;

data = d;

}

};

class SinglyLinkedList {

public:

Node \* head;

SinglyLinkedList() {

head = NULL;

}

SinglyLinkedList(Node \* n) {

head = n;

}

// 1. CHeck if node exists using key value

Node \* nodeExists(int k) {

Node \* temp = NULL;

Node \* ptr = head;

while (ptr != NULL) {

if (ptr -> key == k) {

temp = ptr;

}

ptr = ptr -> next;

}

return temp;

}

// 2. Append a node to the list

void appendNode(Node \* n) {

if (nodeExists(n -> key) != NULL) {

cout << "Node Already exists with key value : " << n -> key << ". Append another node with different Key value" << endl;

} else {

if (head == NULL) {

head = n;

cout << "Node Appended" << endl;

} else {

Node \* ptr = head;

while (ptr -> next != NULL) {

ptr = ptr -> next;

}

ptr -> next = n;

cout << "Node Appended" << endl;

}

}

}

// 3. Prepend Node - Attach a node at the start

void prependNode(Node \* n) {

if (nodeExists(n -> key) != NULL) {

cout << "Node Already exists with key value : " << n -> key << ". Append another node with different Key value" << endl;

} else {

n -> next = head;

head = n;

cout << "Node Prepended" << endl;

}

}

// 4. Insert a Node after a particular node in the list

void insertNodeAfter(int k, Node \* n) {

Node \* ptr = nodeExists(k);

if (ptr == NULL) {

cout << "No node exists with key value: " << k << endl;

} else {

if (nodeExists(n -> key) != NULL) {

cout << "Node Already exists with key value : " << n -> key << ". Append another node with different Key value" << endl;

} else {

n -> next = ptr -> next;

ptr -> next = n;

cout << "Node Inserted" << endl;

}

}

}

// 5. Delete node by unique key

void deleteNodeByKey(int k) {

if (head == NULL) {

cout << "Singly Linked List already Empty. Cant delete" << endl;

} else if (head != NULL) {

if (head -> key == k) {

head = head -> next;

cout << "Node UNLINKED with keys value : " << k << endl;

} else {

Node \* temp = NULL;

Node \* prevptr = head;

Node \* currentptr = head -> next;

while (currentptr != NULL) {

if (currentptr -> key == k) {

temp = currentptr;

currentptr = NULL;

} else {

prevptr = prevptr -> next;

currentptr = currentptr -> next;

}

}

if (temp != NULL) {

prevptr -> next = temp -> next;

cout << "Node UNLINKED with keys value : " << k << endl;

} else {

cout << "Node Doesn't exist with key value : " << k << endl;

}

}

}

}

// 6th update node

void updateNodeByKey(int k, int d) {

Node \* ptr = nodeExists(k);

if (ptr != NULL) {

ptr -> data = d;

cout << "Node Data Updated Successfully" << endl;

} else {

cout << "Node Doesn't exist with key value : " << k << endl;

}

}

// 7th printing

void printList() {

if (head == NULL) {

cout << "No Nodes in Singly Linked List";

} else {

cout << endl << "Singly Linked List Values : ";

Node \* temp = head;

while (temp != NULL) {

cout << "(" << temp -> key << "," << temp -> data << ") --> ";

temp = temp -> next;

}

}

}

};

int main() {

SinglyLinkedList s;

int option;

int key1, k1, data1;

cout << "\n Select Option number. Enter 0 to exit." << endl;

cout << "1. appendNode()" << endl;

cout << "2. prependNode()" << endl;

cout << "3. insertNodeAfter()" << endl;

cout << "4. deleteNodeByKey()" << endl;

cout << "5. updateNodeByKey()" << endl;

cout << "6. print()" << endl;

cout << "7. Clear Screen" << endl << endl;

do {

cout<<"Enter your choice : "<<endl;

cin >> option;

Node \* n1 = new Node();

//Node n1;

switch (option) {

case 0:

break;

case 1:

cout << "Append Node Operation \nEnter key & data of the Node to be Appended" << endl;

cin >> key1;

cin >> data1;

n1 -> key = key1;

n1 -> data = data1;

s.appendNode(n1);

//cout<<n1.key<<" = "<<n1.data<<endl;

break;

case 2:

cout << "Prepend Node Operation \nEnter key & data of the Node to be Prepended" << endl;

cin >> key1;

cin >> data1;

n1 -> key = key1;

n1 -> data = data1;

s.prependNode(n1);

break;

case 3:

cout << "Insert Node After Operation \nEnter key of existing Node after which you want to Insert this New node: " << endl;

cin >> k1;

cout << "Enter key & data of the New Node first: " << endl;

cin >> key1;

cin >> data1;

n1 -> key = key1;

n1 -> data = data1;

s.insertNodeAfter(k1, n1);

break;

case 4:

cout << "Delete Node By Key Operation - \nEnter key of the Node to be deleted: " << endl;

cin >> k1;

s.deleteNodeByKey(k1);

break;

case 5:

cout << "Update Node By Key Operation - \nEnter key & NEW data to be updated" << endl;

cin >> key1;

cin >> data1;

s.updateNodeByKey(key1, data1);

break;

case 6:

s.printList();

cout<<endl;

break;

case 7:

system("cls");

break;

default:

cout << "Enter Proper Option number " << endl;

}

} while (option != 0);

return 0;

}

**Doubly linked list**

#include<iostream>

using namespace std;

class Node {

public:

int key;

int data;

Node \* next;

Node \* previous;

Node() {

key = 0;

data = 0;

next = NULL;

previous = NULL;

}

Node(int k, int d) {

key = k;

data = d;

}

};

class DoublyLinkedList {

public:

Node \* head;

DoublyLinkedList() {

head = NULL;

}

DoublyLinkedList(Node \* n) {

head = n;

}

// 1. CHeck if node exists using key value

Node \* nodeExists(int k) {

Node \* temp = NULL;

Node \* ptr = head;

while (ptr != NULL) {

if (ptr -> key == k) {

temp = ptr;

}

ptr = ptr -> next;

}

return temp;

}

// 2. Append a node to the list

void appendNode(Node \* n) {

if (nodeExists(n -> key) != NULL) {

cout << "Node Already exists with key value : " << n -> key << ". Append another node with different Key value" << endl;

} else {

if (head == NULL) {

head = n;

cout << "Node Appended as Head Node" << endl;

} else {

Node \* ptr = head;

while (ptr -> next != NULL) {

ptr = ptr -> next;

}

ptr -> next = n;

n -> previous = ptr;

cout << "Node Appended" << endl;

}

}

}

// 3. Prepend Node - Attach a node at the start

void prependNode(Node \* n) {

if (nodeExists(n -> key) != NULL) {

cout << "Node Already exists with key value : " << n -> key << ". Append another node with different Key value" << endl;

} else {

if (head == NULL) {

head = n;

cout << "Node Prepended as Head Node" << endl;

} else {

head -> previous = n;

n -> next = head;

head = n;

cout << "Node Prepended" << endl;

}

}

}

// 4. Insert a Node after a particular node in the list

void insertNodeAfter(int k, Node \* n) {

Node \* ptr = nodeExists(k);

if (ptr == NULL) {

cout << "No node exists with key value: " << k << endl;

} else {

if (nodeExists(n -> key) != NULL) {

cout << "Node Already exists with key value : " << n -> key << ". Append another node with different Key value" << endl;

} else {

Node \* nextNode = ptr -> next;

// inserting at the end

if (nextNode == NULL) {

ptr -> next = n;

n -> previous = ptr;

cout << "Node Inserted at the END" << endl;

}

//inserting in between

else {

n -> next = nextNode;

nextNode -> previous = n;

n -> previous = ptr;

ptr -> next = n;

cout << "Node Inserted in Between" << endl;

}

}

}

}

// 5. Delete node by unique key. Basically De-Link not delete

void deleteNodeByKey(int k) {

Node \* ptr = nodeExists(k);

if (ptr == NULL) {

cout << "No node exists with key value: " << k << endl;

} else {

if (head -> key == k) {

head = head -> next;

cout << "Node UNLINKED with keys value : " << k << endl;

} else {

Node \* nextNode = ptr -> next;

Node \* prevNode = ptr -> previous;

// deleting at the end

if (nextNode == NULL) {

prevNode -> next = NULL;

cout << "Node Deleted at the END" << endl;

}

//deleting in between

else {

prevNode -> next = nextNode;

nextNode -> previous = prevNode;

cout << "Node Deleted in Between" << endl;

}

}

}

}

// 6th update node

void updateNodeByKey(int k, int d) {

Node \* ptr = nodeExists(k);

if (ptr != NULL) {

ptr -> data = d;

cout << "Node Data Updated Successfully" << endl;

} else {

cout << "Node Doesn't exist with key value : " << k << endl;

}

}

// 7th printing

void printList() {

if (head == NULL) {

cout << "No Nodes in Doubly Linked List";

} else {

cout << endl << "Doubly Linked List Values : ";

Node \* temp = head;

while (temp != NULL) {

cout << "(" << temp -> key << "," << temp -> data << ") <--> ";

temp = temp -> next;

}

}

}

};

int main() {

DoublyLinkedList obj;

int option;

int key1, k1, data1;

cout << "\nWhat operation do you want to perform? Select Option number. Enter 0 to exit." << endl;

cout << "1. appendNode()" << endl;

cout << "2. prependNode()" << endl;

cout << "3. insertNodeAfter()" << endl;

cout << "4. deleteNodeByKey()" << endl;

cout << "5. updateNodeByKey()" << endl;

cout << "6. print()" << endl;

cout << "7. Clear Screen" << endl << endl;

do {

cout<<"Enter your choice :";

cin >> option;

Node \* n1 = new Node();

//Node n1;

switch (option) {

case 0:

break;

case 1:

cout << "Append Node Operation \nEnter key & data of the Node to be Appended" << endl;

cin >> key1;

cin >> data1;

n1 -> key = key1;

n1 -> data = data1;

obj.appendNode(n1);

//cout<<n1.key<<" = "<<n1.data<<endl;

break;

case 2:

cout << "Prepend Node Operation \nEnter key & data of the Node to be Prepended" << endl;

cin >> key1;

cin >> data1;

n1 -> key = key1;

n1 -> data = data1;

obj.prependNode(n1);

break;

case 3:

cout << "Insert Node After Operation \nEnter key of existing Node after which you want to Insert this New node: " << endl;

cin >> k1;

cout << "Enter key & data of the New Node first: " << endl;

cin >> key1;

cin >> data1;

n1 -> key = key1;

n1 -> data = data1;

obj.insertNodeAfter(k1, n1);

break;

case 4:

cout << "Delete Node By Key Operation - \nEnter key of the Node to be deleted: " << endl;

cin >> k1;

obj.deleteNodeByKey(k1);

break;

case 5:

cout << "Update Node By Key Operation - \nEnter key & NEW data to be updated" << endl;

cin >> key1;

cin >> data1;

obj.updateNodeByKey(key1, data1);

break;

case 6:

obj.printList();

cout<<endl;

break;

case 7:

system("cls");

break;

default:

cout << "Enter Proper Option number " << endl;

}

} while (option != 0);

return 0;

}

**Circular Linked List**

#include<iostream>

using namespace std;

class Node {

public:

int key;

int data;

Node \* next;

Node() {

key = 0;

data = 0;

next = NULL;

}

Node(int k, int d) {

key = k;

data = d;

}

};

class CircularLinkedList {

public:

Node \* head;

CircularLinkedList() {

head = NULL;

}

// 1. CHeck if node exists using key value

Node \* nodeExists(int k) {

Node \* temp = NULL;

Node \* ptr = head;

if (ptr == NULL) {

return temp;

} else {

do {

if (ptr -> key == k) {

temp = ptr;

}

ptr = ptr -> next;

} while (ptr != head);

return temp;

}

//return temp;

}

// 2. Append a node to the list

void appendNode(Node \* new\_node) {

if (nodeExists(new\_node -> key) != NULL) {

cout << "Node Already exists with key value : " <<

new\_node -> key <<

". Append another node with different Key value" <<

endl;

} else {

if (head == NULL) {

head = new\_node;

new\_node -> next = head;

cout << "Node Appended at first Head position" << endl;

} else {

Node \* ptr = head;

while (ptr -> next != head) {

ptr = ptr -> next;

}

ptr -> next = new\_node;

new\_node -> next = head;

cout << "Node Appended" << endl;

}

}

}

// 3. Prepend Node - Attach a node at the start

void prependNode(Node \* new\_node) {

if (nodeExists(new\_node -> key) != NULL) {

cout << "Node Already exists with key value : " <<

new\_node -> key <<

". Append another node with different Key value" <<

endl;

} else {

if (head == NULL) {

head = new\_node;

new\_node -> next = head;

cout << "Node Prepended at first Head position" << endl;

} else {

Node \* ptr = head;

while (ptr -> next != head) {

ptr = ptr -> next;

}

ptr -> next = new\_node;

new\_node -> next = head;

head = new\_node;

cout << "Node Prepended" << endl;

}

}

}

// 4. Insert a Node after a particular node in the list

void insertNodeAfter(int k, Node \* new\_node) {

Node \* ptr = nodeExists(k);

if (ptr == NULL) {

cout << "No node exists with key value OF: " << k << endl;

} else {

if (nodeExists(new\_node -> key) != NULL) {

cout << "Node Already exists with key value : " <<

new\_node -> key <<

". Append another node with different Key value" <<

endl;

} else {

if (ptr -> next == head) {

new\_node -> next = head;

ptr -> next = new\_node;

cout << "Node Inserted at the End" << endl;

} else {

new\_node -> next = ptr -> next;

ptr -> next = new\_node;

cout << "Node Inserted in between" << endl;

}

}

}

}

// 5. Delete node by unique key

void deleteNodeByKey(int k) {

Node \* ptr = nodeExists(k);

if (ptr == NULL) {

cout << "No node exists with key value OF : " << k <<

endl;

} else {

if (ptr == head) {

if (head -> next == NULL) {

head = NULL;

cout << "Head node Unlinked... List Empty";

} else {

Node \* ptr1 = head;

while (ptr1 -> next != head) {

ptr1 = ptr1 -> next;

}

ptr1 -> next = head -> next;

head = head -> next;

cout << "Node UNLINKED with keys value : " << k << endl;

}

} else {

Node \* temp = NULL;

Node \* prevptr = head;

Node \* currentptr = head -> next;

while (currentptr != NULL) {

if (currentptr -> key == k) {

temp = currentptr;

currentptr = NULL;

} else {

prevptr = prevptr -> next;

currentptr = currentptr -> next;

}

}

prevptr -> next = temp -> next;

cout << "Node UNLINKED with keys value : " << k << endl;

}

}

}

// 6th update node

void updateNodeByKey(int k, int new\_data) {

Node \* ptr = nodeExists(k);

if (ptr != NULL) {

ptr -> data = new\_data;

cout << "Node Data Updated Successfully" << endl;

} else {

cout << "Node Doesn't exist with key value : " << k << endl;

}

}

// 7th printing

void printList() {

if (head == NULL) {

cout << "No Nodes in Circular Linked List";

} else {

cout << endl << "head address : " << head << endl;

cout << "Circular Linked List Values : " << endl;

Node \* temp = head;

do {

cout << "(" << temp -> key << "," << temp -> data << "," << temp -> next << ") --> ";

temp = temp -> next;

} while (temp != head);

}

}

};

int main() {

CircularLinkedList obj;

int option;

int key1, k1, data1;

cout << "\nWhat operation do you want to perform? Select Option number. Enter 0 to exit." << endl;

cout << "1. appendNode()" << endl;

cout << "2. prependNode()" << endl;

cout << "3. insertNodeAfter()" << endl;

cout << "4. deleteNodeByKey()" << endl;

cout << "5. updateNodeByKey()" << endl;

cout << "6. print()" << endl;

cout << "7. Clear Screen" << endl << endl;

do {

cout<<"Enter your choice : ";

cin >> option;

Node \* n1 = new Node();

//Node n1;

switch (option) {

case 0:

break;

case 1:

cout << "Append Node Operation \nEnter key & data of the Node to be Appended" << endl;

cin >> key1;

cin >> data1;

n1 -> key = key1;

n1 -> data = data1;

obj.appendNode(n1);

//cout<<n1.key<<" = "<<n1.data<<endl;

break;

case 2:

cout << "Prepend Node Operation \nEnter key & data of the Node to be Prepended" << endl;

cin >> key1;

cin >> data1;

n1 -> key = key1;

n1 -> data = data1;

obj.prependNode(n1);

break;

case 3:

cout << "Insert Node After Operation \nEnter key of existing Node after which you want to Insert this New node: " << endl;

cin >> k1;

cout << "Enter key & data of the New Node first: " << endl;

cin >> key1;

cin >> data1;

n1 -> key = key1;

n1 -> data = data1;

obj.insertNodeAfter(k1, n1);

break;

case 4:

cout << "Delete Node By Key Operation - \nEnter key of the Node to be deleted: " << endl;

cin >> k1;

obj.deleteNodeByKey(k1);

break;

case 5:

cout << "Update Node By Key Operation - \nEnter key & NEW data to be updated" << endl;

cin >> key1;

cin >> data1;

obj.updateNodeByKey(key1, data1);

break;

case 6:

obj.printList();

cout<<endl;

break;

case 7:

system("cls");

break;

default:

cout << "Enter Proper Option number " << endl;

}

} while (option != 0);

return 0;

}

**Tree traversal techniques**

#include<stdio.h>

#include<stdlib.h>

struct node

{

int data;

struct node \*left;

struct node \*right;

};

struct node \*root = NULL;

struct node \*getNode(int val)

{

struct node \*newNode;

newNode = malloc(sizeof(struct node));

newNode->data = val;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

struct node \*insertNode(struct node \*root, int val)

{

if(root == NULL)

return getNode(val);

if(root->data < val)

root->right = insertNode(root->right,val);

if(root->data > val)

root->left = insertNode(root->left,val);

return root;

}

int tree\_height(struct node\* root) {

// Get the height of the tree

if (!root)

return 0;

else {

// Find the height of both subtrees

// and use the larger one

int left\_height = tree\_height(root->left);

int right\_height = tree\_height(root->right);

if (left\_height >= right\_height)

return left\_height + 1;

else

return right\_height + 1;

}

}

void print\_level(struct node\* root, int level\_no) {

// Prints the nodes in the tree

// having a level = level\_no

// We have a auxiliary root node

// for printing the root of every

// subtree

if (!root)

return;

if (level\_no == 0) {

// We are at the top of a subtree

// So print the auxiliary root node

printf("%d -> ", root->data);

}

else {

// Make the auxiliary root node to

// be the left and right nodes for

// the subtrees and decrease level by 1, since

// you are moving from top to bottom

print\_level(root->left, level\_no - 1);

print\_level(root->right, level\_no - 1);

}

}

void print\_tree\_level\_order(struct node\* root) {

if (!root)

return;

int height = tree\_height(root);

for (int i=0; i<height; i++) {

printf("Level %d: ", i);

print\_level(root, i);

printf("\n");

}

printf("\n\n-----Complete Level Order Traversal:-----\n");

for (int i=0; i<height; i++) {

print\_level(root, i);

}

printf("\n");

}

void inorder(struct node \*root)

{

if(root == NULL)

return;

inorder(root->left);

printf("%d ",root->data);

inorder(root->right);

}

void preorder(struct node \*root)

{

if(root == NULL)

return;

printf("%d ",root->data);

preorder(root->left);

preorder(root->right);

}

void postorder(struct node \*root)

{

if(root == NULL)

return;

postorder(root->left);

postorder(root->right);

printf("%d ",root->data);

}

int main()

{

struct node \*root = NULL;

int data,ch;

printf("\n1.To insert a new node.");

printf("\n2.Inorder Traversal.");

printf("\n3.Preorder Traversal.");

printf("\n4.Postorder Traversal.");

printf("\n5.Exit");

do

{

printf("\nSelect one of the operations:");

scanf("%d",&ch);

switch (ch)

{

case 1 :

printf("\nEnter the value to be inserted\n");

scanf("%d",&data);

root = insertNode(root,data);

break;

case 2 :

printf("\nInorder Traversal of the Binary Tree:");

inorder(root);

break;

case 3 :

printf("\nPreorder Traversal of the Binary Tree:");

preorder(root);

break;

case 4 :

printf("\nPostorder Traversal of the Binary Tree:");

postorder(root);

break;

case 5:printf("\nLevel order Traversal of the Binary Tree:");

print\_tree\_level\_order(root);

break;

case 6:

printf("Thank You...");

break;

default :

printf("Wrong Choice\n");

break;

}

} while(ch != 6);

return 0;

}

**Graph Traversal**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_VERTICES 100

int visited[MAX\_VERTICES];

struct Node {

int vertex;

struct Node\* next;

};

struct Graph {

int numVertices;

struct Node\*\* adjLists;

};

void addEdge(struct Graph\* graph, int src, int dest) {

struct Node\* newNode = (struct Node\*) malloc(sizeof(struct Node));

newNode->vertex = dest;

newNode->next = graph->adjLists[src];

graph->adjLists[src] = newNode;

}

void dfs(struct Graph\* graph, int vertex) {

visited[vertex] = 1;

printf("%d ", vertex);

struct Node\* adjList = graph->adjLists[vertex];

while (adjList != NULL) {

int connectedVertex = adjList->vertex;

if (!visited[connectedVertex]) {

dfs(graph, connectedVertex);

}

adjList = adjList->next;

}

}

void bfs(struct Graph\* graph, int startVertex) {

visited[startVertex] = 1;

int queue[MAX\_VERTICES];

int front = -1;

int rear = -1;

queue[++rear] = startVertex;

while (front != rear) {

int currentVertex = queue[++front];

printf("%d ", currentVertex);

struct Node\* adjList = graph->adjLists[currentVertex];

while (adjList != NULL) {

int connectedVertex = adjList->vertex;

if (!visited[connectedVertex]) {

visited[connectedVertex] = 1;

queue[++rear] = connectedVertex;

}

adjList = adjList->next;

}

}

}

int main() {

struct Graph\* graph = (struct Graph\*) malloc(sizeof(struct Graph));

printf("Enter the number of vertices: ");

scanf("%d", &graph->numVertices);

graph->adjLists = (struct Node\*) malloc(graph->numVertices sizeof(struct Node\*));

int i;

for (i = 0; i < graph->numVertices; i++) {

graph->adjLists[i] = NULL;

}

int choice = 0;

int src, dest;

do {

printf("\n1. Add edge\n");

printf("2. BFS traversal\n");

printf("3. DFS traversal\n");

printf("4. Quit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter source and destination vertex: ");

scanf("%d %d", &src, &dest);

addEdge(graph, src, dest);

break;

case 2:

for (i = 0; i < graph->numVertices; i++) {

visited[i] = 0;

}

printf("Enter starting vertex: ");

scanf("%d", &src);

bfs(graph, src);

break;

case 3:

for (i = 0; i < graph->numVertices; i++) {

visited[i] = 0;

}

printf("Enter starting vertex: ");

scanf("%d", &src);

dfs(graph, src);

break;

case 4:

printf("thank you");

exit(0);

default:

printf("Invalid choice\n");

}

}while(choice!=4);

return 0;

}

**Dijkstra’s algorithm to obtain shortest paths**

#include <iostream>

#include <vector>

#include <queue>

#include <limits.h>

using namespace std;

// Graph class

class Graph {

int num\_vertices;

vector<pair<int, int>>\* adj\_list;

public:

// Constructor

Graph(int V) {

num\_vertices = V;

adj\_list = new vector<pair<int, int>>[V];

}

// Method to add edge

void addEdge(int u, int v, int weight) {

adj\_list[u].push\_back(make\_pair(v, weight));

adj\_list[v].push\_back(make\_pair(u, weight));

}

// Dijkstra's algorithm method

void dijkstra(int start\_vertex) {

priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

vector<int> distance(num\_vertices, INT\_MAX);

pq.push(make\_pair(0, start\_vertex));

distance[start\_vertex] = 0;

while (!pq.empty()) {

int u = pq.top().second;

pq.pop();

for (auto i = adj\_list[u].begin(); i != adj\_list[u].end(); i++) {

int v = (\*i).first;

int weight = (\*i).second;

if (distance[v] > distance[u] + weight) {

distance[v] = distance[u] + weight;

pq.push(make\_pair(distance[v], v));

}

}

}

cout << "Shortest distances from vertex " << start\_vertex << ":" << endl;

for (int i = 0; i < num\_vertices; i++) {

cout << i << ": " << distance[i] << endl;

}

}

};

int main() {

int num\_vertices, num\_edges;

cout << "Enter the number of vertices: ";

cin >> num\_vertices;

Graph graph(num\_vertices);

cout << "Enter the number of edges: ";

cin >> num\_edges;

for (int i = 0; i < num\_edges; i++) {

int u, v, weight;

cout << "Enter edge " << i+1 << " (u v weight): ";

cin >> u >> v >> weight;

graph.addEdge(u, v, weight);

}

int start\_vertex;

cout << "Enter the starting vertex: ";

cin >> start\_vertex;

graph.dijkstra(start\_vertex);

return 0;

}

**Binary search using divide and conquer**

#include <iostream>

using namespace std;

int binarySearch(int arr[], int left, int right, int target) {

while (left <= right) {

int mid = left + (right - left) / 2;

if (arr[mid] == target) {

return mid;

} else if (arr[mid] < target) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return -1;

}

int main() {

int size;

cout << "Enter the size of the array: ";

cin >> size;

int arr[size];

cout << "Enter the array elements in sorted order: ";

for (int i = 0; i < size; i++) {

cin >> arr[i];

}

int target;

cout << "Enter the target element to search: ";

cin >> target;

int result = binarySearch(arr, 0, size - 1, target);

if (result == -1) {

cout << "Target element is not present in the array." << endl;

} else {

cout << "Target element is present at index " << result << " in the array." << endl;

}

return 0;

}

**Sorting algorithm using divide and conquer technique**

#include <iostream>

#include <vector>

using namespace std;

// Function prototypes

void mergeSort(vector<int>& arr, int low, int high);

void quickSort(vector<int>& arr, int low, int high);

int partition(vector<int>& arr, int low, int high);

void heapSort(vector<int>& arr, int n);

void heapify(vector<int>& arr, int n, int i);

void printArray(vector<int>& arr);

// Main function

int main() {

int n, choice;

cout << "Enter the number of elements: ";

cin >> n;

vector<int> arr(n);

cout << "Enter the elements:\n";

for(int i=0; i<n; i++) {

cin >> arr[i];

}

cout << "Enter your choice of sorting algorithm:\n";

cout << "1. Merge Sort\n2. Quick Sort\n3. Heap Sort\n";

cin >> choice;

switch(choice) {

case 1:

mergeSort(arr, 0, n-1);

break;

case 2:

quickSort(arr, 0, n-1);

break;

case 3:

heapSort(arr, n);

break;

default:

cout << "Invalid choice!\n";

return 0;

}

cout << "Sorted array:\n";

printArray(arr);

return 0;

}

// Function to perform merge sort

void mergeSort(vector<int>& arr, int low, int high) {

if(low < high) {

int mid = (low + high) / 2;

mergeSort(arr, low, mid);

mergeSort(arr, mid+1, high);

// Merge the sorted sub-arrays

vector<int> temp(high-low+1);

int i = low, j = mid+1, k = 0;

while(i <= mid && j <= high) {

if(arr[i] <= arr[j]) {

temp[k] = arr[i];

i++;

} else {

temp[k] = arr[j];

j++;

}

k++;

}

while(i <= mid) {

temp[k] = arr[i];

i++;

k++;

}

while(j <= high) {

temp[k] = arr[j];

j++;

k++;

}

for(int p=0; p<k; p++) {

arr[low+p] = temp[p];

}

}

}

// Function to perform quick sort

void quickSort(vector<int>& arr, int low, int high) {

if(low < high) {

int p = partition(arr, low, high);

quickSort(arr, low, p-1);

quickSort(arr, p+1, high);

}

}

// Function to perform partition operation in quick sort

int partition(vector<int>& arr, int low, int high) {

int pivot = arr[high];

int i = low - 1;

for(int j=low; j<high; j++) {

if(arr[j] <= pivot) {

i++;

swap(arr[i], arr[j]);

}

}

swap(arr[i+1], arr[high]);

return i+1;

}

// Function to perform heap sort

void heapSort(vector<int>& arr, int n) {

// Build heap

for(int i=n/2-1; i>=0; i--) {

heapify(arr, n, i);

}

// Extract elements one by one from

for(int i=n-1; i>=0; i--) {

swap(arr[0], arr[i]);

heapify(arr, i, 0);

}

}

// Function to heapify a subtree rooted at node i

void heapify(vector<int>& arr, int n, int i) {

int largest = i; // Initialize largest as root

int l = 2\*i + 1; // Left child

int r = 2\*i + 2; // Right child

// If left child is larger than root

if(l < n && arr[l] > arr[largest]) {

largest = l;

}

// If right child is larger than largest so far

if(r < n && arr[r] > arr[largest]) {

largest = r;

}

// If largest is not root

if(largest != i) {

swap(arr[i], arr[largest]);

heapify(arr, n, largest);

}

}

// Function to print the elements of an array

void printArray(vector<int>& arr) {

for(int i=0; i<arr.size(); i++) {

cout << arr[i] << " ";

}

cout << endl;

}

**Knapsack using greedy technique**

#include<iostream>

#include<algorithm>

using namespace std;

// Define a struct to represent an item in the knapsack

struct Item {

int value;

int weight;

};

// Define a comparison function to sort items based on value-to-weight ratios

bool compare(Item a, Item b) {

double ratio1 = (double)a.value / (double)a.weight;

double ratio2 = (double)b.value / (double)b.weight;

return ratio1 > ratio2; // sort in descending order

}

// Define the function to solve the fractional knapsack problem

double fractionalKnapsack(int W, Item arr[], int n) {

// Sort the items in descending order based on value-to-weight ratios

sort(arr, arr + n, compare);

// Initialize the current weight and final value of items in the knapsack

double currentWeight = 0.0;

double finalValue = 0.0;

// Iterate through the items and add them to the knapsack as long as their weight does not exceed the remaining capacity

for (int i = 0; i < n; i++) {

if (currentWeight + arr[i].weight <= W) {

// Add the entire item to the knapsack if its weight does not exceed the remaining capacity

currentWeight += arr[i].weight;

finalValue += arr[i].value;

}

else {

// Add a fraction of the item to the knapsack if its weight exceeds the remaining capacity

int remainingWeight = W - currentWeight;

finalValue += arr[i].value \* ((double)remainingWeight / (double)arr[i].weight);

break; // exit the loop

}

}

// Return the final value of items in the knapsack

return finalValue;

}

// Main function to take input from the user and call the fractionalKnapsack function

int main() {

int n, W;

cout << "Enter the number of items: ";

cin >> n;

cout << "Enter the maximum weight of knapsack: ";

cin >> W;

// Create an array of items and take input from the user for each item's value and weight

Item arr[n];

for (int i = 0; i < n; i++) {

cout << "Enter the value and weight of item " << i+1 << ": ";

cin >> arr[i].value >> arr[i].weight;

}

// Call the fractionalKnapsack function to calculate the maximum value of items that can be put in the knapsack

double maxVal = fractionalKnapsack(W, arr, n);

cout << "The maximum value of items that can be put in the knapsack is: " << maxVal << endl;

return 0;

}

**Travelling Salesman**

#include<iostream>

#include<vector>

#include<cstring>

using namespace std;

#define MAX 9999

int n;

int distan[10][10];

int completed\_visit;

int DP[16][4];

int TSP(int mark,int position){

if(mark==completed\_visit){

return distan[position][0];

}

if(DP[mark][position]!=-1){

return DP[mark][position];

}

int answer = MAX;

for(int city=0;city<n;city++){

if((mark&(1<<city))==0){

int newAnswer = distan[position][city] + TSP( mark|(1<<city),city);

answer = min(answer, newAnswer);

}

}

return DP[mark][position] = answer;

}

int main(){

cout << "Enter the number of places you want to visit: ";

cin >> n;

completed\_visit = (1<<n) -1;

cout << "Enter the distance matrix:\n";

for(int i=0;i<n;i++){

for(int j=0;j<n;j++){

cin >> distan[i][j];

}

}

memset(DP, -1, sizeof(DP));

cout<<"Minimum Distance Travelled by you is "<<TSP(1,0);

return 0;

}

**Queens**

#include<iostream>

#include<vector>

using namespace std;

bool isSafe(vector<vector<int>>& board, int row, int col) {

// Check if the column on this row has already been occupied

for(int i = 0; i < row; i++) {

if(board[i][col] == 1)

return false;

}

// Check if the upper left diagonal has any queen already

for(int i = row, j = col; i >= 0 && j >= 0; i--, j--) {

if(board[i][j] == 1)

return false;

}

// Check if the upper right diagonal has any queen already

for(int i = row, j = col; i >= 0 && j < board.size(); i--, j++) {

if(board[i][j] == 1)

return false;

}

return true;

}

bool solve(vector<vector<int>>& board, int row) {

// Base case: if all rows are filled, return true

if(row == board.size())

return true;

// Recursive case: try placing a queen on each column of this row

for(int col = 0; col < board.size(); col++) {

if(isSafe(board, row, col)) {

board[row][col] = 1; // place the queen

// Recursively try to place the remaining queens on the next rows

if(solve(board, row+1))

return true;

// Backtrack: remove the queen from this cell and try the next cell

board[row][col] = 0;

}

}

// If we have tried all columns on this row and couldn't place a queen, return false

return false;

}

int main() {

int n;

cout << "Enter the size of the chessboard: ";

cin >> n;

// Initialize an n x n chessboard with all cells set to 0

vector<vector<int>> board(n, vector<int>(n, 0));

if(solve(board, 0)) {

// If a solution is found, print the board

cout << "Solution found:" << endl;

for(int i = 0; i < n; i++) {

for(int j = 0; j < n; j++) {

cout << board[i][j] << " ";

}

cout << endl;

}

}

else {

// If no solution is found, print a message

cout << "No solution found." << endl;

}

return 0;

}